2011-05

A fast segmentation algorithm for piecewise polynomial numeric function generators

Frenzen, Christopher L.

J.T. Butler, C. L. Frenzen, N. Macaria, and T. Sasao, "A fast segmentation algorithm for piecewise polynomial numeric function generators," Journal of Computational and Applied

A fast segmentation algorithm for piecewise polynomial numeric function generators

Jon T. Butler[a*] Christopher L. Frenzen[b], Njuguna Macaria[c], and Tsutomu Sasao[d]

[a]Department of Electrical and Computer Engineering, Naval Postgraduate School, Monterey, CA, 93943-5121, U.S.A., jon_butler@msn.com.

[b]Department of Applied Mathematics, Naval Postgraduate School, Monterey, CA, 93943-5216, U.S.A., cfrenzen@nps.edu.

[c]Department of Electrical and Computer Engineering, Naval Postgraduate School, Monterey, CA, 93943-5121, U.S.A., nmacaria@nps.edu.

[d]Department of Computer Science and Electronics, Kyushu Institute of Technology, Iizuka, 820-8502, JAPAN, sasao@cse.kyutech.ac.jp.

Abstract

We give an efficient algorithm for partitioning the domain of a numeric function f into segments. The function f is realized as a polynomial in each segment, and a lookup table stores the coefficients of the polynomial. Such an algorithm is an essential part of the design of lookup table methods [5,8,9,12,14,15] for realizing numeric functions, such as $sin(\pi x)$, $ln(x)$, and $\sqrt{-ln(x)}$. Our algorithm requires many fewer steps than a previous algorithm given in [6] and makes tractable the design of numeric function generators based on table lookup for *high-accuracy* applications. We show that an estimate of segment width based on local derivatives greatly reduces the search needed to determine the exact segment width. We apply the new algorithm to a suite of 15 numeric functions and show that the estimates are sufficiently accurate to produce a minimum or near-minimum number of computational steps.

Keywords: piecewise linear approximation; numeric function generators; segmentation algorithm

1. Introduction

The existence of large logic circuits has led to increased interest in an old problem - the realization of numeric functions. More than 150 years ago, Babbage designed the difference engine to automatically compute logarithmic and trigonometric functions [1]. This was intended to replace hand computation which was prone to error.

The availability of circuits to compute quickly functions like $sin(x)$ and $log(x)$ offers real-time execution of algorithms that can be used in applications such as the rendering

*Corresponding author at: Department of Electrical and Computer Engineering, Naval Postgraduate School, Code EC/Bu, Monterey, CA, 93943-5121.

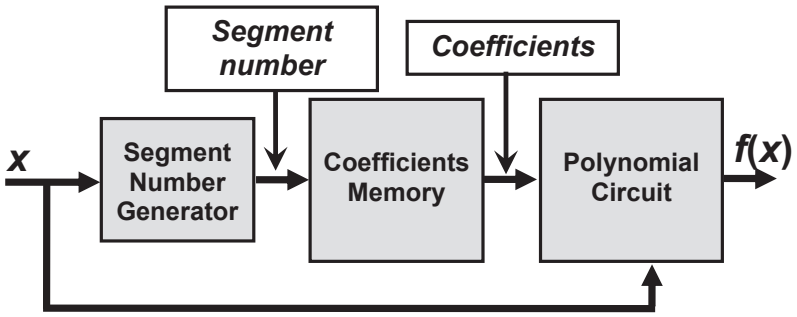

Figure 1. Architecture of a numerical function generator using a piecewise polynomial approximation.

of graphics or digital signal processing.

In this paper we give a new, efficient algorithm for partitioning the domain of a numeric function f into segments. Within each segment, the function f is realized as a polynomial with a lookup table storing the coefficients of the polynomial. We use an estimate of segment width based on local derivatives to greatly reduce the search needed to determine the exact optimal segment width. We then apply our new algorithm to a suite of 15 numeric functions, showing that the estimates are sufficiently accurate to produce a minimum or near-minimum number of steps in the computation.

We note that lookup tables have been used previously to implement a truncated series expression approximation of the given function. Hassler and Takagi [7] represent the function by a converging series and replace the single large memory by two or more smaller lookup tables. Stine and Schulte [16,17] use the Taylor series expansion of a differentiable function. The first two terms of the expansion are realized by using small lookup tables. The reciprocal, square root, inverse square root, and certain elementary functions were realized by Ercegovac, Lang, Muller, and Tisserand [5] using a Taylor expansion and tables. Lookup tables have been used in the implementation of logarithm and antilogarithm computations by Paul, Jayakumar, and Khatri [14].

Lee, Luk, Villasenor, and Cheung [8,9] realize trigonometric and logarithmic functions by table lookup using a non-uniform segmentation method. In their algorithm, narrow segments are used where the change in the function is large, and wide segments are used where the change in the function is small.

Sasao, Butler, and Riedel [15] use the Douglas-Peucker [4] algorithm to partition a given function into segments that are realized by a linear approximation. They show that a circuit producing a non-uniform segmentation has a tractable realization for common numeric functions. Unfortunately, the Douglas-Peucker algorithm does not produce optimum segmentations [6].

Fig. 1 shows the architecture of the numeric function generator (NFG) that realizes a given function as a piecewise polynomial approximation [11]. It consists of three blocks. The Segment Number Generator uses the value of x to generate a segment number that is applied to the address input of the Coefficients Memory. The Coefficients Memory produces the coefficients in the polynomial expression for the given function. The piecewise polynomial approximation $f(x) \approx c_m x^m + \cdots + c_1 x + c_0$ is computed by the Polynomial Circuit in Fig. 1 using the coefficients produced by the Coefficients Memory.

Each segment in the function domain corresponds to a word in the memory which stores the polynomial coefficients for the function approximation in that segment. For a given approximation error, we seek a segmentation of the domain that has the fewest segments possible. This minimizes the memory required for the lookup table.

The algorithm given in this paper efficiently divides the domain of f into segments so that the the error in polynomial approximation in each segment is no greater than a specified error. In [6], we presented an algorithm that produced a segmentation with the fewest segments. However, this algorithm can be computationally intensive, with a computation time sometimes measured in days or weeks. Although applied only once in the synthesis of a numeric function generator, the previous algorithm can make high accuracy applications impractical. Our main result, the new algorithm presented here, is orders of magnitude faster and still yields the fewest steps.

While the proposed segmentation algorithm applies to any order approximating polynomial, our experimental results focus on linear and quadratic approximations. Nagayama, Sasao, and Butler [12] show that presently available field programmable gate arrays (FPGAs) have insufficient arithmetic elements, such as multipliers, to efficiently implement third or higher order polynomials. As FPGA technology improves, this may change.

2. Background

Because the variable x and the function's value $f(x)$ are represented as binary numbers with a fixed number of bits, a numeric function generator's output is inherently an approximation of the exact function value. While we may view the value of x as exact, it may not be possible to view $f(x)$ as exact. For example, consider the function $f(x) = \sqrt{x}$. If $x = 2$, we can realize 2 exactly. However, the irrationality of $\sqrt{2}$ means that its exact value cannot be realized in finitely many bits.

3. Estimating the Segment Width

An essential part of the new segmentation algorithm is deriving an estimate of the segment width. An accurate estimate is essential, because subsequently a search must be performed for the exact segment width. Later, we analyze the estimate's accuracy and show that, in many cases, it is as accurate as it can possibly be. First, we focus on deriving the estimate.

Let the segment over which we seek an nth-order polynomial approximation span $[e, s]$. The maximum approximation error ε of a Chebyshev approximation [10] is

$$\varepsilon = \frac{2(e-s)^{n+1}}{4^{n+1}(n+1)!} \max_{s \le x \le e} |f^{(n+1)}(x)|. \tag{1}$$

Solving (1) for the segment width, $e - s$ yields

$$e - s = 4 \sqrt[n+1]{\frac{(n+1)!\varepsilon}{2 \max_{s \le x \le e} |f^{(n+1)}(x)|}}. \tag{2}$$

For the two special cases of linear and quadratic approximating polynomials, we have

$$e - s|_{\text{linear}} = 4\sqrt{\frac{\varepsilon}{\max_{s\le x\le e}|f''(x)|}} = 4\sqrt{\frac{\varepsilon}{|f''_{e-s}(x*)|}} \tag{3}$$

and

$$e - s|_{\text{quadratic}} = 4\sqrt[3]{\frac{3\varepsilon}{\max_{s\le x\le e}|f'''(x)|}} = 4\sqrt[3]{\frac{3\varepsilon}{|f'''_{e-s}(x*)|}}, \tag{4}$$

where $e-s|_{\text{linear}}$ and $e-s|_{\text{quadratic}}$ are the segment widths for linear and quadratic approximations, respectively. We have chosen to replace $\max_{s\le x\le e}|f^{(n+1)}(x)|$ and $\max_{s\le x\le e}|f'''(x)|$ by the abbreviations $|f''_{e-s}(x^*)|$ and $|f'''_{e-s}(x^*)|$, respectively, recognizing that if the appropriate derivative is continuous on a closed interval, then the maxima above will each be attained at some point x^* within that interval.

4. The Segmentation Algorithm

4.1. Introduction

The algorithm is shown in Table 1. It applies to polynomial approximations of any order. We assume that the function domain is represented by a vector of N discrete points. For example, if the interval is $[0, 1)$ and the accuracy is 8 bits, then we may choose N to be 256, and the points, in binary to be $0.0000\ 0000_2$, $0.0000\ 0001_2$, $0.0000\ 0010_2$, ... , and $0.1111\ 1111_2$. That is, the domain in this example is the vector $[0.000, 0.0039, 0.0078, \ldots, 0.9961]$. This assumption is consistent with the algorithm's implementation in MATLAB [3]. In MATLAB, we associate this vector with variable x. $f(x)$, in MATLAB, is then a vector of elements corresponding to the function evaluated at each of the elements in x. Therefore, $f(x)$ also has N elements.

Definition 1. *For a given function, a* **step** *in a segmentation algorithm is a computation of the maximum absolute error between the function and its approximating polynomial on the proposed segment.*

Because so much computation time occurs in the calculation of the maximum absolute error, a step, as defined in Definition 1, is an appropriate measure of the execution time. We compare the number of steps needed in the proposed algorithm with the number of steps needed in a brute force method.

In the brute force segmentation, the beginning point of the first segment is chosen to be the leftmost point in the interval of approximation; i.e. x_{low}. Then, the second point and successive points are chosen as prospective end points, and, for each choice, the error between the function and its approximating polynomial is computed. When this error exceeds the approximation error ε, the exact segment width has been found; it corresponds to the end point just before the end point that resulted in an error that exceeded ε. In the brute force method, all but the leftmost point in the interval is a prospective end point at which the error between the function and its approximating

Table 1
Algorithm to segment a given function based on estimates of the segment length

Algorithm 1: Segment the domain $[x_{\text{low}}, x_{\text{high}}]$ of a given function $f(x)$, where $f(x)$ is approximated in each segment by a polynomial $c_n x^n + \cdots + c_1 x + c_0$.
Input: Function $f(x)$, domain $[x_{\text{low}}, x_{\text{high}}]$, approximation error ε, and order of the approximating polynomial, n. **Output:** Optimum segmentation, in which the i-th segment is specified as $[s_i, e_i]$, where s_i and e_i are the beginning and end point, respectively.
1. $i \leftarrow 1$. $s_1 \leftarrow x_{\text{low}}$.
ESTIMATE 2. Estimate the current segment width determining end point e_{est} and approximation error ε_{est}. If $e_{\text{est}} > x_{\text{high}}$, then $e_{\text{est}} \leftarrow x_{\text{high}}$. If $e_{\text{est}} = x_{\text{high}}$ and $\varepsilon_{\text{est}} \leq \varepsilon$, then STOP with $e_i \leftarrow e_{\text{est}}$.
LOCATE 4. If $\varepsilon_{\text{est}} < \varepsilon$, then find upper and lower bounds H and L on the segment end point with the property a) $\varepsilon_L \leq \varepsilon < \varepsilon_H$, where ε_H and ε_L are the approximation errors for the segments $[s_i, H]$ and $[s_i, L]$, respectively. Go to Step 5. b) $\varepsilon_H \leq \varepsilon$ and $H = x_{\text{high}}$. STOP with $e_i \leftarrow e_{\text{est}}$. If $\varepsilon_{\text{est}} \geq \varepsilon$, then find upper and lower bounds H and L on the segment end point.
PINPOINT 5. Using H and L, produce Hpp and Lpp with the property $\varepsilon_{Lpp} \leq \varepsilon < \varepsilon_{Hpp}$, where ε_{Hpp} and ε_{Lpp} are the approximation errors for the segments $[s_i, Hpp]$ and $[s_i, Lpp]$, respectively that are *adjacent* points above and below the optimum segment width. Choose the segment end point e_i to be Lpp.
6. $s_{i+1} \leftarrow$ point above e_i. $i \leftarrow i + 1$. Go to Step 2.

polynomial is computed. Therefore, approximately N steps are needed, where N is the number of points to represent the function in the whole interval of approximation.

The algorithm proceeds from the smallest value in the domain x_{low} to the largest x_{high}. It establishes the largest segment, starting at x_{low}, such that the maximum approximation error is ε. It repeats this process starting at e_1, the end point of the first segment, until it reaches x_{high}. Often, the last segment is truncated because x_{high} is reached before a segment end occurs (where the approximation error is ε). As a result, it is not unusual for the last segment to have a maximum approximation error strictly less than ε.

Fig. 2 shows an example segmentation. The vertical axis plots the function value $f(x)$, while the horizontal axis plots x. x_{low} is the left-hand end of the interval over which $f(x)$ is realized, and x_{high} is the right-hand end of the interval.

4.2. Three Parts to the Algorithm

There are three parts to the algorithm.

In the first part, ESTIMATE, the segment width is estimated. The process of estimation

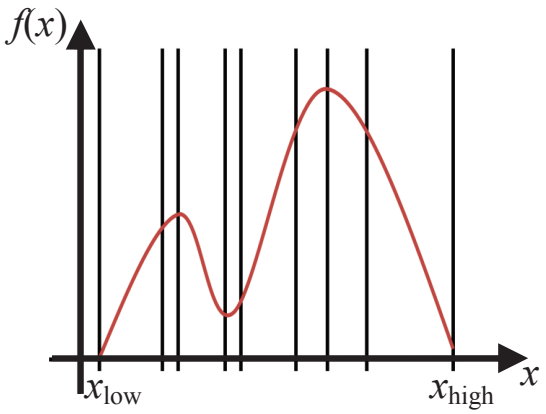

Figure 2. Example segmentation.

is discussed in the next section. Using the estimated segment width, an end point is found and the approximation error for the proposed segment is computed. This counts as one step.

In the second part, LOCATE, two points in the domain are located such that one point yields a segment whose approximation error is just below (or equal to) ε, and the other point yields a segment whose approximation error is just above.

This is accomplished as follows: from the estimated segment width computed in ESTIMATE, it is known whether the corresponding point is above the optimum segment width or below (or equal). If above, LOCATE proceeds towards lower values of x searching for two points that straddle the optimum segment width. If below, LOCATE proceeds towards higher values. Assume the point is below. The algorithm proceeds toward the optimum segment width by one point initially. It computes the approximation error of the new segment, adding 1 to the number of steps. If the approximation error exceeds ε, ESTIMATE stops. It has found two points on each side of the optimum segment width. Indeed, they are adjacent and the algorithm stops; there is no need to proceed to the next step, PINPOINT.

However, if the approximation error is still less than (or equal to) ε, the algorithm advances *two* points. Again, it computes the approximation error, adding 1 to the number of steps, and repeats the process above. The is repeated with the algorithm advancing four, eight, etc. points, until two points are found that are on each side of the optimum segment width. If a total of m steps are taken, the algorithm has advanced $1+2+4+\ldots+2^{m-1} = 2^m - 1$ points. At the end of ESTIMATE, the last two points considered, H and L, correspond to end points of segments that straddle the *exact* end point of the segment Specifically, H is the end point of a segment in which the error achieved is either greater than ε, and L is the end point of a segment in which the error achieved is less than or equal to ε.

In the third part, PINPOINT, a bisection method is applied to H and L. That is, the midpoint $A = \frac{H+L}{2}$ is computed. Then, a new segment whose end point is A is created, and its approximation error is computed. If this exceeds ε, then H is replaced by A and the process is repeated. If this is less than or equal to ε, then L is replaced by A and the process is repeated. Each time a new approximation error is computed, the number of steps is increased by 1. The process stops when the H and L are adjacent. The segment

end point is chosen to be L, since the maximum error in the segment ending in L is less than or equal to ε, while the maximum error in the segment ending in H exceeds ε.

4.3. Number of Steps

Because of the way Algorithm 1 is constructed, the difference between H and L is 2^m. We have the following lemma.

Lemma 1. *Let the number of points between the high and low point $H - L$ be a power of 2, 2^m. For all but the last segment, the average and the worst case number of steps $N_{\text{PINPOINT}}(m)$ required by PINPOINT is m. No steps are required by PINPOINT to compute the last segment.*

Proof The proof is by induction on m. For $m = 0$, H and L are adjacent, and no further steps are needed. Assume the hypothesis is true for all $m < m'$, and consider $H - L = 2^{m'}$. There is one step required to compute the approximation error for a segment that ends at $P = \frac{H+L}{2}$. Either H or L is replaced by P, and the problem is one of determining the number of steps needed to compute the segment end point between (the new) H and L. Since $H - L = 2^{m'-1}$, from the assumption, $m' - 1$ steps are needed, for a total of m' steps.

No steps are required by PINPOINT to compute the last segment because H is x_{high} and the error associated with a segment end point of H is equal to or less than ε.

∎

Similarly, the number of steps required by LOCATE can be calculated, as shown in Lemma 2. We begin with a definition.

Definition 2. *A* **truncated segment** *is a segment whose estimated end point is greater than x_{high}.*

For each segment, the algorithm in Table 1 provides an estimated segment end point that is used to start the search for the exact end point. A truncated segment has the property that its estimated end point is greater than x_{high}. Often, a truncated segment occurs as the last (rightmost) segment in a segmentation.

Interestingly, a truncated segment is not necessarily the last segment. For example, suppose that in a linear approximation of the function, the function is nearly linear throughout most of the interval, except near the end. In this case, the segment's proposed end point may reach the end point of the interval of approximation (especially, if the segment is near the interval end point). Thus, it may be a truncated segment. However, when PINPOINT is applied, the exact end point may be found to be an internal point. Since the next segment might be in a highly non-linear part of the domain, the segment is necessarily narrow, and its end point may not reach the interval's end point. Therefore, subsequently constructed segments may be *non*-truncated. In the course of generating the experimental data, our proposed algorithm encountered this phenomena.

Lemma 2. *The number of steps required to construct a non-truncated segment in LOCATE, $N_{\text{LOCATE}}(m)$, is $N_{\text{PINPOINT}}(m) + 2$.*

Proof The proof is by induction on m. For $m = 0$, H and L are adjacent, and PINPOINT requires no steps. The approximation error associated with segments whose end points are H and L require a total of two steps. Therefore, $N_{\text{PINPOINT}}(0) = 0$ and $N_{\text{LOCATE}}(0) = 2$. Assume the hypothesis is true for all $m < m'$, and consider m'. It follows that $N_{\text{LOCATE}}(m') = N_{\text{LOCATE}}(m' - 1) + 1$. The hypothesis follows.

∎

For the last segment, LOCATE requires some number of steps before it is determined that $H = x_{\text{high}}$. At this point, if the approximation error with H as the segment end point is equal to or less than ε, then it is established that, indeed, this is the last segment. No steps are needed by PINPOINT.

In the best case, ESTIMATE produces a segment end point that is no more than one step away from the optimum segment end point. This requires one step. To verify this and thus terminate the segment construction, another step is required, for a total of two steps per segment. From the discussion above, a truncated segment may, in the best case, require only one step. Therefore, we have

Lemma 3. *At least* $2s - 1$ *steps are needed to segment a domain, where* s *is the number of segments in the segmentation.*

Lemma 3 assumes that the estimates of segment length are as accurate as possible. As N, the number of points in the domain, becomes large, then the percentage of steps needed compared to the brute force method approaches 0. The program shows a clear tendency to lower percentage of steps as N increases.

5. Artifacts Associated With the Use of Different Accuracies

5.1. A Conundrum

Intuition suggests that using many points (e.g. 10,000,000) to represent an interval of approximation $[x_{\text{low}}, x_{\text{high}}]$ yields a more accurate segmentation than when fewer points are used (e.g. 256). Thus, one expects the segments to be narrower (or the same) when fewer points are used. On the contrary, if the segments are wider, the approximation error will be greater than ε. Therefore, one expects *more* segments (or the same) are needed when there are fewer points to represent the interval of approximation.

However, this is not the case. Table 2 shows the number of segments needed to realize three functions, $\sqrt{-ln(x)}$, $-(x \log_2 x + (1 - x) \log_2(1 - x))$, and $sin(e^x)$, using 8-bits of precision and a linear approximation[2]. There are two cases, $N = 256$ and $N = 10,000,000$. For all three functions, the number of segments for $N = 10,000,000$ is *larger* than for $N = 256$.

5.2. Resolution

In the algorithm shown in Table 1, the beginning point of a segment is the next point after the end point of the previous segment (*not* the same point at which the last segment ends). This recognizes that each point belongs to exactly one segment. Thus, there is "space" between segments which need not be realized by a polynomial. This is benign; there are no input combinations that correspond to values in this space. For example, in the case of $sin(e^x)$, there are 27 segments, and thus, 26 spaces between segments. Since

[2] $\sqrt{-ln(x)}$, $-(x \log_2 x + (1–x) \log_2(1–x))$, and $sin(e^x)$ were considered in [8], [6], and [11], respectively.

Table 2
Three functions which require fewer segments when $N = 256$ than when $N = 10,000,000$ for a linear piecewise approximation.

Function $f(x)$	**Interval** x	**No. of Segs.** $N = 256$	$N = 10^7$
$\sqrt{-ln(x)}$	$[\frac{1}{256}, \frac{1}{4})$	12	14
$-(x \log_2 x + (1-x) \log_2(1-x))$	$(0, 1)$	19	20
$sin(e^x)$	$[0, 2]$	27	28

only 256 points represent the segment, more than 26/256 of the interval is *not* realized in the approximation. This effectively shortens the interval by about 10%. In the case of $N = 10,000,000$, the space between segments is a much smaller fraction of the total interval width. This effect dominates and is the reason that fewer points yields fewer segments in Table 1.

6. Experimental Results

6.1. Benefits of Estimates

To analyze the benefit of estimates in the proposed algorithm, we configured a MATLAB program to apply only LOCATE and PINPOINT in constructing each segment. That is, estimates were *not* used in specifying a prospective end point of the next segment. Instead, the initial end point was chosen to be just beyond the beginning point of the newly constructed segment. In this case, LOCATE and PINPOINT must search over the full segment. Table 3 shows how this compares to the brute force method when applied to a suite of 15 functions for $\varepsilon = 2^{-17}$. Each entry represents the ratio of the number of steps needed to compute the segmentation using the proposed algorithm to the number of steps needed by the brute force method. This is expressed as a percentage. The values, shown in the column labeled **# of Estimates = 0**, range from 0.7244% for $\frac{1}{1+e^{-x}}$ to 10.1860% for $sin(e^x)$. This shows that LOCATE and PINPOINT realize a significant reduction over the brute force method. For 13 of the 15 functions, the ratios are less than 5.0%, which is a significant reduction in the number of steps.

However, estimates provide still further improvement. Table 3 shows the benefits of 1, 2, and 3 estimates. The column labeled **# of Estimates = 1** shows that, when one estimate is used, the number of steps is reduced by as much as one-fifth that needed in the case of one estimate. For example, in the case of the entropy function $-(x \log_2 x + (1-x) \log_2(1-x))$ no estimate yields a percentage of 7.7408%, while one estimate achieves a percentage of 1.3785%, which is 1/5.6 of the number of steps.

In the case of one estimate, the beginning point of the segment is used to determine an estimate for the segment width. For example, when linear approximation is used, the second derivative of the new segment beginning point is computed and substituted into (3) to derive an estimate for the segment width. Then, a proposed end point is obtained by adding the estimated segment width to the beginning point. The approximation error is computed and used to determine in which direction from the estimated end point LOCATE should search.

The next column labeled **# of Estimates = 2** shows the benefit of two estimates. In this case, the estimate of the segment width computed with the first step in the segment

Table 3
Percentage of steps (compared to brute force) required to segment functions approximated by linear polynomials using different estimates of segment width for $N = 2^{16}$ and $\varepsilon = 2^{-17}$.

Function $f(x)$	**Interval** x	**% of Steps vs. Brute Force**				**Min. %**	**# of Segs**
		0	**1**	**2**	**3**		
2^x	$[0, 1)$	2.28	0.46	*0.23	*0.23	0.23	75
$1/x$	$[1, 2)$	2.34	0.75	*0.23	*0.23	0.23	75
$\sqrt{x}$	$[1, 2)$	1.19	0.46	*0.11	*0.11	0.11	35
$1/\sqrt{x}$	$[1, 2)$	1.62	0.62	*0.15	*0.15	0.15	50
$\log_2(x)$	$[1, 2)$	2.35	0.67	*0.23	*0.23	0.23	76
$\ln x$	$[1, 2)$	2.00	0.60	*0.19	*0.19	0.19	63
$sin(\pi x)$	$[0, \frac{1}{2})$	3.16	0.71	0.38	0.35	0.33	109
$cos(\pi x)$	$[0, \frac{1}{2})$	3.15	0.70	0.35	*0.33	0.33	109
$tan(\pi x)$	$[0, \frac{1}{4})$	2.25	0.83	0.27	0.25	0.22	73
$\sqrt{-ln(x)}$	$[\frac{1}{256}, \frac{1}{4}]$	4.87	1.36	*0.63	*0.63	0.63	207
$tan^2(\pi x) + 1$	$[0, \frac{1}{4})$	4.25	0.82	*0.46	*0.46	0.46	152
$-(x \log_2 x + (1-x) \log_2(1-x))$	$[\frac{1}{256}, \frac{255}{256}]$	7.74	1.38	*0.96	*0.96	0.96	314
$\frac{1}{1+e^{-x}}$	$[0, 1)$	0.72	0.37	0.14	0.10	0.06	20
$\frac{1}{\sqrt{2\pi}} e^{\frac{-x^2}{2}}$	$[0, \sqrt{2}]$	2.32	0.84	0.38	0.30	0.23	53
$sin(e^x)$	$[0, 2)$	10.19	2.05	1.43	1.40	1.35	449

* *All* segments require the fewest steps.

(discussed in the previous paragraph) is averaged with the estimate of the segment width computed with the segment end point, as estimated from the first step. This approach is based on the assumption that the average of two estimates, one at the beginning and one near the end of the proposed segment provides a better estimate of the actual segment width than one estimate alone. As can be seen in Table 3, two estimates provides substantial reduction in the number of steps. Indeed, for 9 of the 15 functions, the minimum number of steps is achieved (where the minimum was *not* achieved for any of the 15 function in the case of one estimate). An asterisk indicates that this percentage is the best that can be obtained, as shown in Lemma 3. The reduction in the number of steps achieved by using two estimates instead of one ranges from 1/1.4 to 1/4.4.

The next column labeled **# of Estimates = 3** shows the benefit of three estimates. In this case, the final estimate is the average of three estimates, one from the beginning, one from the end, and one from the middle of the segment whose width is estimated from the first point in the segment. Now, 10 of the 15 functions achieve the minimum number of steps.

The column labeled **Min %** shows a percentage that represents the minimum number of steps required if the estimates were perfect, as specified by Lemma 3. Comparing this with the column labeled **# of Estimates = 3** shows that, even for the five functions that

did not achieve a minimum number of steps, the number of steps is close to minimum. Four of the five functions are within 30%, while one $\frac{1}{1+e^{-x}}$ is within 72%.

Table 4 shows the detail of a segmentation of the domain of two functions, $sin(\pi x)$ and $cos(\pi x)$ using 8 bit precision with a given approximation error of 2^{-9}. Specifically, this shows the number of steps in each of the seven segments required to compute the segment width. It also shows the number of steps per segment if the brute force method was used. As discussed earlier, this tracks the segment width. Note that the number of steps per segment in the $sin(\pi x)$ function is monotone decreasing. This is because the segment width decreases with increasing x, as $sin(\pi x)$ becomes increasingly nonlinear. In a similar way, the number of steps in the $cos(\pi x)$ function is monotone increasing. It is interesting that the number of steps to compute the segmentation for $sin(\pi x)$ is much greater than for $cos(\pi x)$. For $\varepsilon = 2^{-9}$, $sin(\pi x)$ requires 20.4% of the total number of steps required by the brute force method, while $cos(\pi x)$, requires only 6.5%. Indeed, most of the additional steps in $sin(\pi x)$ over $cos(\pi x)$ occur in the first segment.

Table 4
Number of steps needed to segment $sin(\pi x)$ and $cos(\pi x)$ using linear approximation.*

Function	**% Steps of Brute Force**	**Segment Number**						
		1	**2**	**3**	**4**	**5**	**6**	**7**
$sin(\pi x)$	20.4%	>16,440	<2,184	<893	<481	<277	<149	>1
# Steps/Seg.		25,133	15,963	13,617	12,451	11,790	11,424	9,622
$cos(\pi x)$	6.5%	>26	>120	>238	>417	>754	>1,677	>3,265
# Steps/Seg.		11,280	11,463	11,865	12,582	13,858	16,526	22,426

*For 8 bit precision, $\varepsilon = 2^{-9}$ and $N = 100,000$. The function is approximated by a piecewise linear polynomial using a segment width estimate from the beginning of the segment.

The > and < show the direction the algorithm had to go to achieve the optimum segmentation. For example, the entry $> 16,440$ in the column **Segment Number = 1** and the row $sin(\pi x)$ means the estimate was short for the leftmost segment in the segmentation of $sin(\pi x)$, and it was necessary to *increase* x to achieve an optimum segment. It used 16,440 steps in the algorithm.

Both $sin(\pi x)$ and $cos(\pi x)$ require the same number of segments, as would be expected. However, there is a significant difference in the number of steps. This is because the algorithm begins at lower values of x, where the second derivative of $sin(\pi x)$ is 0 and the second derivative of $cos(\pi x)$ is π^2 (a linear approximation is used). When the second derivative is 0, the estimate of segment width associated with this is infinite. In the algorithm, a large segment width is substituted whose value is computed from the smallest non-zero value of the second derivative (over the whole interval of approximation). From the data, it is clear that this estimate is far away from the actual segment width. Consequently, many steps are needed to achieve the optimum segment width. In subsequent segments, the estimate is more accurate and significantly fewer steps are needed. In the case of $sin(\pi x)$, the region of small second derivative occurs after four or five segments have been established. Indeed, the data indicates that the last segment width for $cos(\pi x)$

is longer than needed, in which case, a segment width that takes the segmentation to the largest x value is taken, and only one step in needed.

Table 4 shows the total number of points in each segment. For example, for Segment 1 of $sin(\pi x)$, 25,133 points occur, which means the brute force method would require 25,133 steps. Because of the estimate, however, only 16,440 steps are needed to establish the segment. Included in this number are 2 steps, one to cross right of the end point, where it is found that ε has been exceeded, and one to cross back.

Note that the estimate for the width of Segment 1 of $cos(\pi x)$ is much better. Only 26 steps are needed to find the segment end point. It is tempting to believe that the number of steps associated with Step i of $sin(\pi x)$ and Step $8 - i$ of $cos(\pi x)$ should be the same, for $1 \leq i \leq 7$. This is not true because Segment 7 for either $sin(\pi x)$ or $cos(\pi x)$ is not a full segment, while all other segments are full segments. That is, the algorithm moves from left to right, and does not need a full segment for Segment 7.

We note that $N = 100,000$ is much larger than one would normally use when $\varepsilon = 2^{-9}$. We have chosen an example with a small number of segments, 7, for clarity's sake and a number of points more representable of a practical segmentation. Also, the function was carefully chosen to illustrate how the number of steps in the algorithm depends on where the function's second derivative is 0.

Table 5 shows the detail of a segmentation for $sin(\pi x)$ and $cos(\pi x)$ using quadratic approximations. Here, 16 bit precision is achieved with a given approximation error of 2^{-17}. In the case of quadratic approximation, the same phenomena seen in linear approximation exists but to a lesser extent. In quadratic approximation, a *third* derivative that is 0 makes the estimate inaccurate versus the second derivative in the case of linear approximation. The roles of the $sin(\pi x)$ and $cos(\pi x)$ reverse, as the third derivative of $cos(\pi x)$ is 0 at $x = 0$. Here, more steps are required for the $cos(\pi x)$ than for the $sin(\pi x)$, but not much more.

Table 5
Number of steps needed to segment $sin(\pi x)$ and $cos(\pi x)$ using quadratic approximation.

Function	**% Steps of Brute Force**	**Segment Number**											
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
$sin(\pi x)$	0.2242%	>6	<8	<10	<12	<12	<14	>14	<16	>16	>18	<20	>1
# Steps/Seg.		4741	4761	4804	4871	4967	5099	5282	5538	5916	6538	7893	5114
$cos(\pi x)$	0.2486%	>22	>20	>18	>16	>16	>14	>14	>12	>12	>10	>8	>1
# Steps/Seg.		9654	6997	6157	5691	5388	5176	5022	4911	4831	4778	4747	2172

*For 17 bit precision, $\varepsilon = 2^{-17}$ and $N = 65,535$. The function is approximated by a piecewise quadratic polynomial using a segment width estimate from the beginning of the segment. The Remez Algorithm was applied once.

6.2. The Benefit of LOCATE and PINPOINT

Table 6 compares the functions $sin(\pi x)$ and $cos(\pi x)$ with respect to the number of steps required in each segment. For both functions, the first row shows the number of steps required by LOCATE to compute each segment. The second row shows the number of steps required by PINPOINT. The third row shows the total of these two. The fourth

row shows the total number of points in each segment, out of a total of $N = 5,000,000$. We have used many more points to represent the interval of approximation than would normally be used when $\varepsilon = 2^{-9}$ to illustrate the case of high precision with a tractable number of segments. The total number of steps in each segment is also the number of steps the brute force method would take to compute that segment. It is noteworthy that Algorithm 1 requires only 0.0028% and 0.0025% of the steps needed by the brute force method for the $sin(\pi x)$ and $cos(\pi x)$ functions, respectively. This also shows the merit of estimates. That is, the use of an estimate reduces the number of steps by roughly one-half.

Note that when an estimate is used, the $sin(\pi x)$ function requires more steps than the $cos(\pi x)$ function. This is due to the inaccuracy of the estimate when the second derivative is 0, which occurs in the first (leftmost) segment of $sin(\pi x)$ and in the last (rightmost) segment of $cos(\pi x)$. Because of this, the first segment of the $sin(\pi x)$ function requires many more steps, 34, than any other segment. Since the same point occurs as the last segment of $cos(\pi x)$, it is a truncated segment, and the inaccuracy has a much less effect on the number of steps needed 1. It follows that the number of steps required by the algorithm is dependent where the second derivative is 0.

Table 6
Number of steps needed to segment $sin(\pi x)$ and $cos(\pi x)$ using estimates from three points*.

Operation	**% Steps of Brute Force**	**Segment Number** 1	2	3	4	5	6	7
No Estimate		$sin(\pi x)$						
LOCATE		22>.	21>	21>	21>	21>	21>	20>
PINPOINT		20>.	19>	19>	19>	19>	19>	0>
TOTAL	0.0052%	42>.	40>	40>	40>	40>	40>	20>
# Steps/Seg.		1,256,637	798,153	680,848	622,519	589,489	571,189	481,165
No Estimate		$cos(\pi x)$						
LOCATE		21>	21>	21>	21>	21>	21>	22>
PINPOINT		19>	19>	19>	19>	19>	19>	0>
TOTAL	0.0052%	40>	40>	40>	40>	40>	40>	22>
# Steps/Seg.		563,989	573,146	593,248	629,071	692,868	826,274	1,121,404
Three Estimates		$sin(\pi x)$						
LOCATE		18<.	13<	12<	11<	11<	11<	1>
PINPOINT		16<.	11<	10<	9<	9<	9<	0>
TOTAL	0.0028%	34<.	24<	22<	20<	20<	20<	1>
# Steps/Seg.		1,256,637	798,153	680,848	622,519	589,489	571,189	481,165
Three Estimates		$cos(\pi x)$						
LOCATE		11<	11<	11<	11<	12<	12<	1>
PINPOINT		9<	9<	9<	9<	10<	10<	0>
TOTAL	0.0025%	20<	20<	20<	20<	22<	22<	1>
# Steps/Seg.		563,989	573,146	593,248	629,071	692,868	826,274	1,121,404

*For $\varepsilon = 2^{-9}$ and $N = 5,000,000$, and estimates from three points.

There is a nearly linear relationship between the number of steps and the number of segments. This can be seen in the scatter plot of the percentage of steps compared to

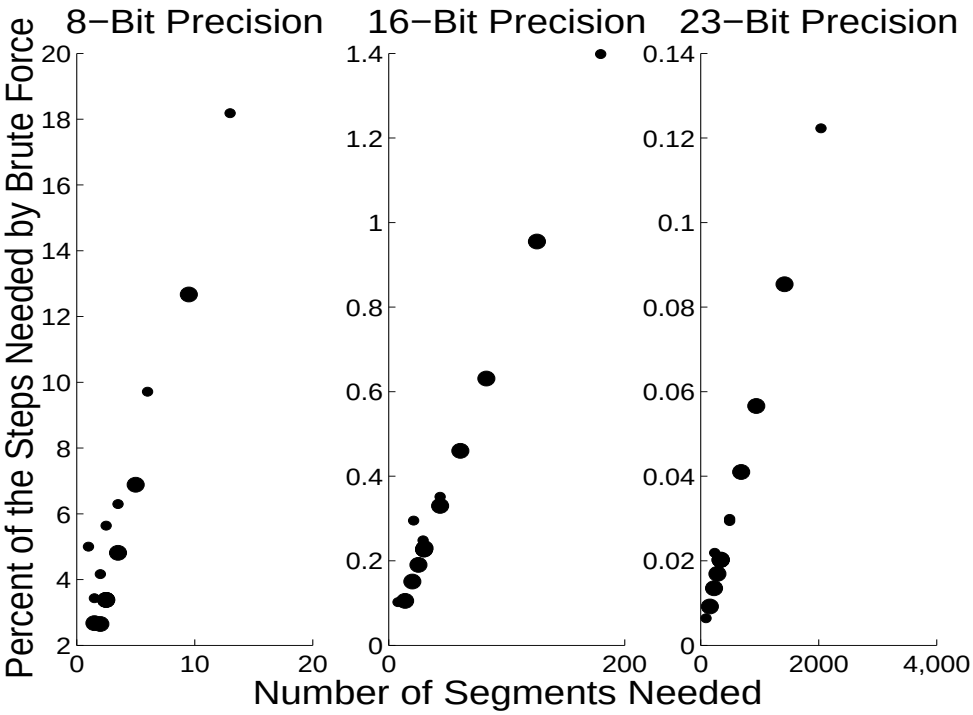

Figure 3. Percent of the steps by brute force versus the number of segments

brute force versus the number of segments of Fig. 3[3]. A reason for this is the fact that the estimates for 16-bit precision are close enough that most functions require the fewest steps per segment, which is 2. For example, the $\frac{1}{\sqrt{x}}$ function requires 50 segments, which is 0.1508% of the steps required by brute force. The total number of steps is 99 ($= 2 \times 49 + 1$), which is the minimum. However, the Gaussian function $\frac{1}{\sqrt{2\pi}} e^{\frac{-x^2}{2}}$ requires 53 segments, which is 0.2950% of the steps required by brute force, slightly less than twice that of the $\frac{1}{\sqrt{x}}$ function. This is because of extra steps needed near $x = 1$, where the function is close to linear, and the estimates are less accurate.

7. Concluding Remarks

We have given a segmentation algorithm that efficiently segments a given numeric function, such as $sin(\pi x)$, in such a way that the polynomial approximation error is less than some given value. The algorithm requires many fewer steps than a previous algorithm [6]. Experimental results show that, in some instances, only the absolute minimum number of steps is needed. In most instances, it requires close to the minimum number of steps.

[3]We chose 23 bit precision instead of 24 bit precision because the fraction in the 32-bit single precision IEEE Floating Point Standard (ANSI/IEEE Std 754-1985) is 23 bits.

8. Acknowledgments

This research is supported in part by an NSA Contract, Grants in Aid for Scientific Research of JSPS, and MEXT, and a grant of the Kitakyushu Innovative Cluster Project.

REFERENCES

1. A. G. Bromley, "The evolution of Babbage's calculating engines," *Annals of the History of Computing*, Vol. 9 pp. 113-136, 1987.
2. D. Cochran, "The Evolution of Babbage's Calculating Engines," *Annals of the History of Computing*, Vol. 9 pp. 113-136, 1987.
3. S. J. Chapman, *MATLAB Programming for Engineers*, Brooks/Cole Thomson Learning, 2nd Edition, p. 52, 2002.
4. D. H. Douglas and T. K. Peucker, "Algorithms for the reduction of the number of points required to represent a line or its caricature," *The Canadian Cartographer*, Vol. 10, No. 2, pp. 112-122, 1973.
5. M. D. Ercepovac, T. Lang, J.-M. Muller, and A. Tisserand, "Reciprocation, square root, inverse square root, and some elementary functions using small multipliers," *IEEE Trans. Comp.*, Vol. 49, No. 7, pp. 628-637, July 2000.
6. C. L. Frenzen, T. Sasao, and J. T. Butler, "On the number of segments needed in a piecewise linear approximation", *Jour. of Computational and Applied Mathematics*. Vol. 234, pp. 437-466, May 2010.
7. H. Hassler and N. Takagi, "Function evaluation by table lookup and addition," *Proc. of the 12th IEEE Symp. on Computer Arithmetic (ARITH'95)*, Bath, England, pp. 10-16, July 1995.
8. D. U. Lee, Wayne, Luk, J. Villasenor, and P. Y. K. Cheung, "Non-uniform segmentation for hardware function evaluation," *Proc. Inter. Conf. on Field Programmable Logic and Applications*, pp. 796-807, Lisbon, Portugal, Sept. 2003
9. D.U. Lee, Wayne, Luk, J. Villasenor, and P. Y. K. Cheung, "A hardware Gaussian noise generator for channel code evaluation," *Proc. of the 11th Annual IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM'03)*, Napa, CA, pp. 69-78, April 2003.
10. J. H. Mathews, *Numerical Methods for Computer Science, Engineering, and Mthematics*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 19877.
11. J. Muller, *Elementary Functions - Algorithms and Implementation*, Birkhäuser, Boston, 1997.
12. S. Nagayama, T. Sasao, and J. T. Butler, "Design method of numerical function generators based on polynomial approximation for FPGA implementation," *10th Euromicro Conference on Digital System Design, Architecture, Methods, and Tools (DSD 2007)*, August 27-31, 2007. Lübeck, Germany.
13. R. Nave, "Implementation of transcendental function on a numerics processor," *Microprocessing and Microprogramming*, Vol. 11, pp. 221-225, 1983.
14. S. Paul, N. Jayakumar, and S. P. Khatri, "A hardware approach for approximate, efficient logarithm and antilogarithm computations," *IWLS-2007*, San Diego, CA, pp. 260-265, May 30-June 1, 2007,
15. T. Sasao, J. T. Butler, and M. D. Riedel, "Application of LUT cascades to numerical function generators," *The 12th Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI2004)*, Kanazawa, Japan, pp. 422-429, Oct. 18-19, 2004.
16. M. J. Schulte and J. E. Stine, "Approximating elementary functions with symmetric bipartite tables," *IEEE Trans. on Computers,*, Vol. 48, No. 8, pp. 842-847, Aug. 1999.
17. J. E. Stine and M. J. Schulte, "The symmetric table addition method for accurate function approximation," *Jour. of VLSI Signal Processing*, Vol. 21, No. 2, pp. 167-177, June, 1999.